You Can't Be Me: Enabling Trusted Paths & User Sub-Origins in Web Browsers

Enrico Budianto[1], Yaoqi Jia[1], Xinshu Dong[2], Prateek Saxena[1], Zhenkai Liang[1],

National University of Singapore[1], Advanced Digital Sciences Center[2]
{enricob,jiayaoqi,prateeks,liangzk}@comp.nus.edu.sg,
xinshu.dong@adsc.com.sg

Abstract. Once a web application authenticates a user, it loosely associates all resources owned by the user to the web session established. Consequently, any scripts injected into the victim web session attain unfettered access to user-owned resources, including scripts that commit malicious activities inside a web application. In this paper, we establish the first explicit notion of *user sub-origins* to defeat such attempts. Based on this notion, we propose a new solution called USERPATH to establish an end-to-end trusted path between web application users and web servers. To evaluate our solution, we implement a prototype in Chromium, and retrofit it to 20 popular web applications. USERPATH reduces the size of client-side TCB that has access to user-owned resources by 8x to 264x, with small developer effort.

Keywords: User sub-origins, trusted path, script injection attacks

1 Introduction

Many of the web applications today, such as DropBox, Gmail and Facebook, provide user-oriented services, where users need to create their own accounts to use the service tailored to them. User-oriented web applications isolate data belonging to individual users and bind access control privileges to specific user accounts (e.g., owners or administrators). In such web applications, the authority of a user is typically represented by a web session, and the security mechanisms are centered on protecting the web session state from being accessed by attackers. In such a setting, if an attacker is able to inject scripts into the session, the scripts run with user's full authority. In this paper, we do not focus on mechanisms to prevent web application vulnerabilities from occurring. Rather, we propose mechanisms to defend against *post-attack* malicious behavior of an injected script, which we term as *post-injection script execution* (PISE) attacks. Our proposal serves as a second line of defense when existing mechanisms of script injection prevention, such as Content Security Policy [1], fail to achieve full coverage [2].

PISE attacks are the aftermath of script-injection attacks that occur in a variety of ways, such as mixed content (over HTTP) in HTTPS sessions [3], loading malicious third-party scripts [4], or via XSS attacks [5]. The threat model in PISE attacks is strong and challenging to counteract: injected scripts already run under the same origin as the web application. In this work, we focus on PISE attacks that target sensitive data owned by users and mimic normal user interactions within a web application. For example, XSS worms on Facebook profiles that utilize self-XSS attacks to befriend certain users [6] or malicious extensions that stealthily steal authentication credentials and hijack user accounts [7] are some of the real-world examples.

We observe two fundamental limitations of the present web platform. First, to defeat PISE attacks, browsers need to have the notion of a *user authority* that controls access to sensitive user-owned resources. The *same-origin policy* does not support such access control. Second, there is no direct way for server-side web applications to be faithfully informed about user's interaction at the client-side. As a result, web servers cannot, for example, distinguish between web requests generated in response to legitimate user interaction versus requests generated by injected scripts, even in the presence of web sessions protection mechanisms like HTTPS. A recent line of research has proposed piecemeal defenses to mitigate some classes of PISE attacks via client-side channels [8, 9], server-side channels [10, 11], self-exfiltration [12], or using attacks that mimic user interactions to legitimize dangerous information flows [13]. However, none of them offer a comprehensive solution to prevent PISE attacks completely.

Our Solution. We propose a solution called USERPATH, which augments the present web platform with a security primitive that explicitly represents a ***User*** authority and establishes an end-to-end *trusted **Path*** between the user and the server. We introduce the first explicit notion of *user sub-origins*[1] into web applications, which are primitives that run with the authority of web application users. Our mechanism enables user sub-origins to isolate user's data and privilege-separate the code operating on it from the rest of the web origin. Thus, our mechanism tightens the authority of the web application users from web sessions to user-suborigins. To support our end-to-end system, we build a trusted path between human users and the web application server [15]. A trusted path in our work is defined as a privileged channel, which allows the server to tightly and reliably control the communication of visible content and input with the user (via the standard DOM APIs), even in the presence of malicious application code. Although this concept has recently been explored to develop new access control mechanisms on mobile and traditional operating systems [15, 16], building it for the web has only recently been investigated [9].

Our solution is easy to deploy in practice – with a small number of changes in existing browsers and web applications, USERPATH can be set up to protect users from PISE attacks. We reuse the existing web isolation primitives and minimize new abstractions added. Our solution is a 475 lines of code patched on Chromium 12. USERPATH-enabled browsers are backward-compatible with non-USERPATH-enabled websites. From the user's perspective, using a USERPATH-enabled website would be largely identical to the original site, except for verifying a colored login input box when authenticating with a password (see Section 4). As a result, USERPATH has a much lower adoption cost as compared to another recent trusted-path proposal that requires generation and uploading of SSL keys for every website [9]. Furthermore, our solution can also be easily deployed with modest development effort. Specifically, developers

[1] Recently, browsers have added support for per-page sub-origins [14] that compartmentalize contents on a web page within several sub-authorities under the same origin. The per-page sub-origin proposal offers no guarantee to defend against PISE attacks, and we complement per-page sub-origins with the additional notion of user authority and trusted path.

can easily retrofit web applications to use USERPATH simply by privilege-separating sensitive data and JavaScript logic on a client-side user-suborigins called UFrame. UFrame is an `iframe`-like component that isolates code under a different JavaScript context and has the ability to render tamper-proof HTML elements. Such privilege separation of JavaScript code is straightforward for developers to use, as argued in recent works [17, 18].

From a security standpoint, users no longer trust a website at the time of login if script injection vulnerabilities are present in the website. Then, how does a user login and setup an authenticated trusted path? We address this critical issue by introducing secure UI elements [16] that protect user's login credential from malicious client-side code and using a PAKE protocol [19]. A PAKE protocol is a *zero knowledge* protocol that lets two parties authenticate each other without revealing secret information (e.g., a password) through the communication channel. Having authenticated the user, USERPATH maintains isolation of sensitive resources throughout the session by resorting to user sub-origins and a trusted path.

Summary of Results. We deploy USERPATH on 20 popular open-source web applications. The evaluation demonstrates that our solution can protect user-owned data from PISE attacks in these applications with modest adoption effort (in the order of days). For each application, we label a number of data fields as sensitive, and modify the application logic to use USERPATH abstractions. We find that USERPATH eliminates the threats to user data from 325 historical security vulnerabilities in these applications, and reduces the trusted computing base (TCB) size by 8x to 264x. Finally, the performance overhead incurred by our solution is negligible for real-world applications. All case studies and the Chromium-based implementation are available online [20], and we release a video demonstrating the smooth user experience with a USERPATH-enabled browser [21].

Contributions. In summary, we make the following contributions in the paper:

- *End-to-end Solution*. Our main contribution lies in analyzing the attack model we term as PISE attacks, examining the various dimensions of attacks, and providing an end-to-end solution to defeat them. We adapt and combine some known techniques with our new ones to achieve a solution that is easy to deploy on the existing web platform. To the best of our knowledge, this is the first comprehensive defense against PISE attacks targeting user-owned resources, which is a significant subset of self-exfiltration attacks [12].
- *User Sub-Origins & Trusted Path*. We propose the first explicit notion of user sub-origins on the web. We further develop an end-to-end trusted path to eliminate PISE attacks targeting user-owned data.

2 Problem Definition

The missing notion of user sub-origins in today's web sessions gives rise to various attacks threatening web applications. We summarize such attacks and elaborate how they can occur in an existing web application.

2.1 PISE Attacks Targeting User-owned Data

Unlike in traditional OSes (e.g., UNIX), there is no built-in notion of a user authority on the present web, where users login into sites and authenticate themselves using custom password-based interfaces. Authentication of subsequent HTTP requests is performed via "bearer tokens", such as session IDs, CSRF tokens, or cookies. In the presence of script injection vulnerabilities, these tokens are prone to attacks, either via direct token stealing [22], phishing attempts [23], or session riding (e.g., fake HTTP request [24]). In this paper, we term such illegitimate accesses from malicious scripts to resources owned by benign victim users as *post-injection script execution* (PISE) attacks.

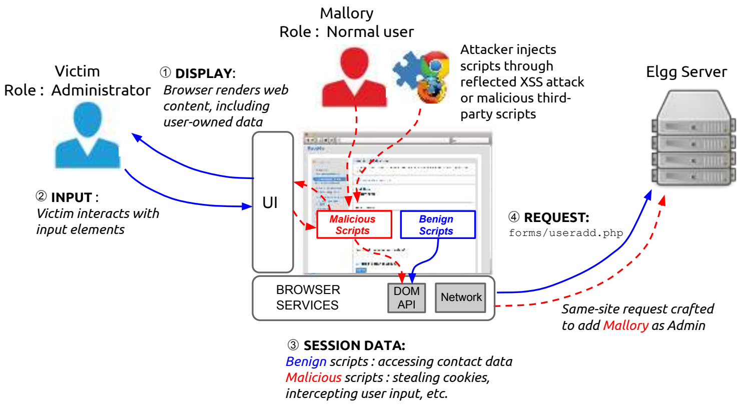

Mallory
Role : Normal user
Attacker injects scripts through reflected XSS attack or malicious third-party scripts
Victim
Role : Administrator
① DISPLAY: Browser renders web content, including user-owned data
Elgg Server
② INPUT : Victim interacts with input elements
UI
Malicious Scripts
Benign Scripts
④ REQUEST: forms/useradd.php
BROWSER SERVICES
DOM API
Network
Same-site request crafted to add Mallory as Admin
③ SESSION DATA:
Benign scripts : accessing contact data
Malicious scripts : stealing cookies, intercepting user input, etc.

Fig. 1: Example Interactions in Elgg. Solid lines illustrate benign interactions between the user, UI elements and session data. Dashed lines illustrate the examples of PISE attacks, where an attacker injects malicious scripts into the victim's session, steals the victim's CSRF token, and performs a same-site request forgery attack to the Elgg server.

We illustrate various PISE attacks with a real-world social networking application called Elgg[2]. Elgg maintains user profiles, manages private message dispatch and blogging, and integrates itself with other social networking sites. Consider the following features available to administrators:

- *Add New User*: This is a privileged feature that can only be accessed by administrators. The administrator specifies information belonging to a particular user that is going to be added to the system. The administrator can also mark the user as a new administrator by identifying it on a checkbox element. Thereafter, this particular information is sent to the server using HTML Form submit mechanism.
- *Profile Management*: Elgg provides profile data management to maintain particular information for each user, similar to most social networking applications. In addition, there is a feature to set other users as administrators directly from their profile pages. However, this feature is privileged to an administrator. The administrator can add another user as an administrator by clicking on "Make admin" link on the user's profile page.

[2] http://elgg.org/

In PISE attacks, injected scripts can access user-owned resources (e.g., the state of "is admin" checkbox of a user) located at the client side and the server side, as shown in Figure 1. We systematically analyze the various channels available to PISE attacks.

At the client side, we categorize three variants of PISE attacks depending on different channels that are exposed to an attacker.

- **Display Channel Attacks.** An attacker can tamper with display elements of a web application to steal sensitive information from users. Two examples of attacks that exploit this channel are UI defacing and phishing for user credentials. In UI defacing attacks, an attacker alters the web content to mislead users. For instance, a malicious extension can change the appearance of a profile page in Facebook [6]. Besides, malicious scripts can also introduce fake UI elements (such as fake login input) to steal users' credentials, therefore allowing them to impersonate as Alice on a site O. Unlike traditional phishing attacks where a malicious website mimics another benign website, in this example the malicious scripts are running within the victim origin O. Therefore, common security indicators such as SSL lock icons and URL bars do not help Alice in detecting the phishing attempt.
- **Input Channel Attacks.** In order to tamper with sensitive data, an attacker can exploit this channel by (1) intercepting or stealing user input; or (2) launching an attack that programmatically interacts with the interface element of the web [13, 16]. In the second scenario, malicious scripts can impersonate a user by forging a user interaction with the DOM element on the web page (e.g., auto-clicking the "add user" button) and mimic the user's action. Another popular attack that exploits this channel (and the display channel) is clickjacking [25], which typically runs in a different website than on Elgg. It can, for instance, load Elgg in a transparent overlay. Then underneath Elgg, it can render another malicious web page to attract users to click on the "Make admin" button in the invisible Elgg layer above. Clickjacking attacks sabotage a user's intention to interact with a UI element as intended by an attacker.
- **Session Data Channel Attacks.** Malicious scripts injected into the web page have access to arbitrary data. It can exfiltrate sensitive data, including cookies, CSRF tokens, capability-bearing URLs, and passwords, through two channels: directly to an attacker-controlled website [8] or via the victim's website itself, which is recently discussed and termed as self-exfiltration attacks by Chen et.al. [12]. Due to lack of input sanitization on Elgg's "edit page" functionality [26], cookie data can be stolen and exfiltrated using XSS attacks via a public blog entry, which is visible to the attacker. This is a confirmed security bug and has been documented as a CVE entry [27].

In addition to these three attack variants, the injected scripts have access to the network, allowing the attacker to access server-side resources of the user.

- **Network Request Channel Attacks.** Malicious scripts can craft and send HTTP requests to the server by invoking `XMLHttpRequest` API, or using HTML's resource tag attributes, such as a `src` attribute in an `<img>` tag. Such crafted requests can be used to perform specific operations on the server-side application. Some websites implement CSRF tokens that are sent along with HTTP requests and server-side applications verify whether the incoming requests carry expected

CSRF tokens. However, secret CSRF tokens and other existing defenses for CSRF attacks, such as `Referer` and `Origin` headers [28], do not suffice for preventing requests forged by PISE attacks, as the injected scripts run in the same origin.

2.2 Insufficiency of Existing Solutions

Many existing solutions provide piecemeal defenses against PISE attacks. In Table 1, we briefly compare existing second line of defense techniques to mitigate this class of attacks. The comparison is categorized based on the four channels exposed to the attackers (Section 2.1). As Table 1 summarizes, none of them provides full protection for the four channels against malicious scripts injected into victim web sessions. We refer readers to Section 6 for a detailed comparison with previous solutions. We propose a user-based end-to-end trusted path that comprehensively protects all the four channels.

Table 1: Various Techniques for Mitigating PISE Attacks

	I[1]	II[2]	III[3]	IV[4]		I[1]	II[2]	III[3]	IV[4]
HTML5 Privilege Separation [18]			√		WebWallet [29]		√		
HTML5 Data Confinement [8]			√	√	Secure UI Toolkit [16]	√	√	√	
Object-Capability Sec Model [30, 31]			√		Clickjacking Defenses [32]	√	√		
PathCutter [24]			√	√	Cryptons [9]		√	√	
Request Triggering Attribution [13]		√		√	DOMinator [33]			√	
Adsentry [34]			√		Origin Bound Certificates [22]			√	
UserPath	√	√	√	√					

[1] Display Channel [2] Input Channel [3] Session Data Channel [4] Network Request Channel

2.3 Threat Model & Scope

We now briefly discuss the in-scope threats of our work. We consider the attacker to be a standard *web attacker* [35] that is able to exploit script injection vulnerabilities in a web application and browser's add-ons running as JavaScript (not binary plugins) [36]. All attacker payloads are client-side scripts, and we assume an uncompromised web server and web browser, as well as the underlying OS. We assume that the user is *benign*, i.e., we do not aim to prevent an attack where an authenticated user attacks the web applications within its own user authority. An HTTP parameter tampering attack, wherein Alice might attack Elgg for profit (e.g., randomly add users to increase number of friends), is such an example [37]. We also assume the security of user passwords, i.e., the users do not disclose their passwords nor use the same password for different websites. Lastly, although our approach is applicable to non-JavaScript-based attacks in concept, our discussion here precludes malicious Flash scripts or Java Applets embedded in web pages.

3 UserPath Design & Security Properties

To protect user-owned resources in the web application from PISE attacks, we combine various techniques to protect the channels exposed to attackers (Section 3.2). Our solution requires minor changes to today's web browsers and web applications, and is easy to use for end users.

3.1 Challenges & Key Ideas

Protection for sensitive user-owned resource should cover the entire life time of web sessions, starting from user authentication to the teardown of the web session. We explain the challenges in doing so below.

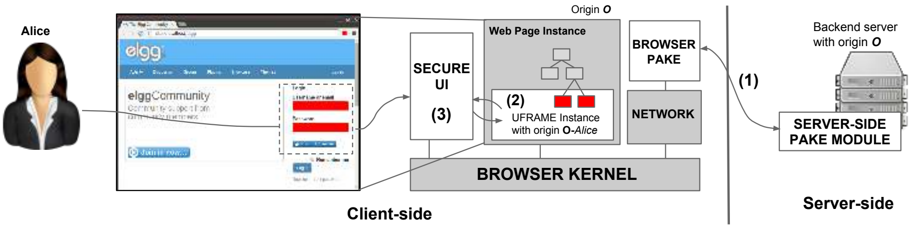


Fig. 2: Overview of USERPATH. The unshaded boxes are the contributions of our paper. A USERPATH-enabled platform has (1) server- and client-side PAKE modules to carry out PAKE protocol, (2) a web primitive called UFrame, and (3) secure UI elements.

Protecting User Credentials. Malicious scripts can exploit display channels to launch in-application phishing attacks and steal the user's password. Note that browser's security indicators (e.g., SSL lock icon, URL bar) do not help users recognize such attacks. Those security indicators operate under the assumption that a web session in an origin is trusted. Such an assumption becomes invalid with our threat model, as the attacks take place within the same session of the victim's origin. To achieve a secure authentication, our idea is to allow a web browser to render secure login elements on the web applications (Section 4). Such elements are special UI controls rendered by the browser, which can be easily verified by the user and cannot be tampered with by untrusted JavaScript code. Once users enter their credentials, leaking these credentials to an untrusted environment (a script or server) is not desirable. To address this critical problem, we employ a PAKE protocol (Figure 2 Step 1) that enables the web browser to authenticate a user to a web origin without directly exchanging credential information with the origin O.

Establishing Notion of User. After the successful authentication, another challenge is to securely establish a notion of user inside a web session. We term this step as *secure delegation* (Section 4), in which the browser creates a user sub-authority in origin O. This step constitutes a form of authority delegation on the web. To achieve this goal, the key idea is to conceptually split the web session into two partitions, one web session running under the authority of the web application origin O, the other one running under a user sub-origin O_{Alice}. USERPATH ties all sensitive resources belonging to user Alice under the sub-origin O_{Alice}, which represents the explicit notion of Alice's *sub-authority*[3] (Figure 2 Step 2). Note that code running in O_{Alice} represents the authority of Alice in O, and is more privileged than the origin O's code.

End-to-End Trusted Path. Fully protecting the four vulnerable channels is challenging with any single mechanism. Instead, we safeguard each vulnerable channel by provid-

[3] This secure delegation process is akin to executing an `su - alice` command in a UNIX-like system.

ing the corresponding secure channel: a secure channel between the UFrame and the backend server, a secure channel between the UFrame and the browser kernel components, a secure visual channel, and a secure input channel – the latter two channels are established with the web application user (Figure 2 Step 3). This constitutes an end-to-end trusted path between the user and the server, as further discussed in Section 3.2.

3.2 USERPATH Design

Protecting User Credentials. To initiate the authentication process, USERPATH leverages the standard authentication mechanism using username and password, which can also be extended for SSO-based authentication (see Section 3.4). The process starts with a user Alice visiting a web page with the origin O. Alice interacts with the application under the authority of its web origin O (Figure 5 Step A). The web application invokes a DOM API to draw a special "credential box" (see Figure 3) for Alice to enter her password. The origin O decides the placement and location of the credential box on the web page and Alice needs nothing more than her usual password for this step. Unlike prevailing password boxes where the input is directly accessible to the web page, the data entered by Alice in the credential box will stay in the memory of the browser and is not accessible by the application code. Therefore, it prevents attacker's scripts from stealing the password. The `url` property of the credential box element identifies the server-side script that handles user login.

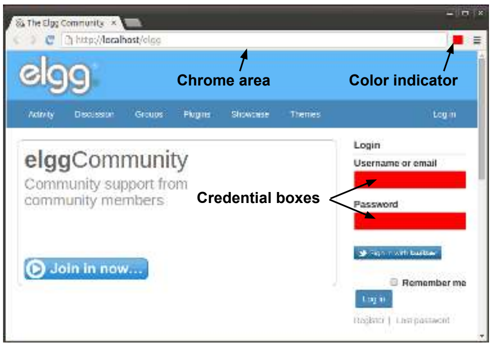


Fig. 3: A web browser displaying credential boxes from `example.com`.

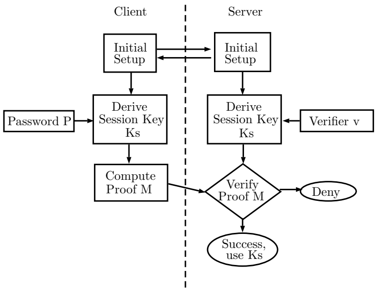


Fig. 4: The PAKE Protocol. A session key S is derived and the server-side PAKE verifies the message M obtained from the client.

After Alice entered her credential information, the browser then executes a PAKE protocol between the browser and the backend server using Alice's password as a secret, without directly exchanging Alice's password with the backend server (Figure 5 Step B). We illustrate the high-level overview of a PAKE protocol in Figure 4. In this protocol, the server O is assumed to have gotten a verifer v which was derived from the Alice's predefined password P. The verifier v is not a password, and cannot be used by Alice for authentication. After Alice enters password P, the client-side PAKE sends Alice's user information and, based on the user information, the server-side PAKE determines the corresponding verifier v. Client-side PAKE (based on user' password) and server-side PAKE (based on verifier v) simultaneously derive a session key K_s, as well as an evidence value M (for client-side PAKE) and M' (for server-side PAKE), according to a set of computations defined in [38]. The message M is later sent by the client to and verified by the server-side PAKE, and vice versa for the message M'. In case of a successful authentication, the common key K_s will be used as a session key for further communications between both parties.

To allow users to distinguish the credential input element drawn by USERPATH from any other similar-looking elements rendered by malicious application code, the browser displays a rectangle of color M in its chrome area and updates the color M simultaneously around the credential box[4]. The user recognizes the authentic credential elements by a visual check. Therefore, this approach defeats any phishing attempts from malicious scripts.

Establishing Notion of User. After authentication is carried out using the PAKE protocol, USERPATH initiates the secure delegation to establish a user sub-authority O_{Alice}. USERPATH creates a UFrame to run Alice's privileged code separated from the rest of the application code within a web origin O. Unlike the temporary origin (e.g., sandboxed `iframe` [18]) which runs in a distinct privileged environment, the UFrame runs within the user Alice's authority with a higher privilege than any other parts in the web page. As a privileged entity, the UFrame has *one-way access* to (1) the main page's DOM via special DOM APIs including access to secure UI elements; (2) a direct secure callback channel to the browser; and (3) a dedicated `XMLHttpRequest` object to make HTTP requests to the backend server. USERPATH privilege-separates user-owned data from being accessed by the less-privileged application code running in O's authority, as well as separates all code that processes user events and the associated user-owned data.

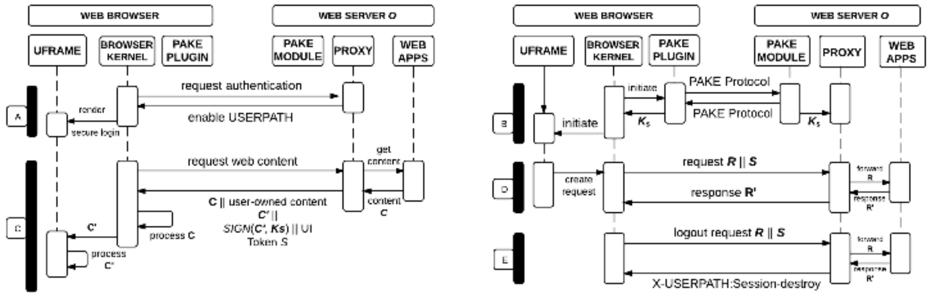


Fig. 5: Sequence of operations in a USERPATH-enabled session.

So far, USERPATH ensures that the sensitive data in UFrame-protected code is not accessible to the less-privileged code (e.g., malicious JS code). But, how to make sure that the UFrame code itself is not initialized with the attacker's payload when it is fetched from the backend server? The UFrame code from the server can be hijacked

[4] The browser dynamically decides a foreground text color in the credential input element that has high contrast with the current background color M and randomizes it every t=5 seconds. To quantitatively measure the entropy, we set M to be randomly chosen from a palette of RGB code colors. This gives a total entropy of 24 bits.

by malicious scripts using a variety of ways, such as DOM clobbering [39] or prototype hijacking of `XMLHttpRequest` object [40]. This lets an attacker create fake UFrames or tamper with the original content of a UFrame. In order to securely delegate user-owned resource to the UFrame, the backend server signs the code with K_s and passes the code for the UFrame to the browser at the initialization step. Once the code is received by the browser, it checks the integrity and authenticity of the code with respect to K_s. Subsequently, the browser bootstraps the UFrame and provides a dedicated `XMLHttpRequest` channel to securely communicate back to the server's origin. At this point, USERPATH has established a secure **UFrame** $\leftrightarrow$ **Server** channel. Note that we consider the server to be uncompromised in our threat model. If a web application developer wishes to isolate users' data better on the server side, several previous solutions such as CLAMP [10] and DIESEL [11] can be used in conjunction with USERPATH's abstractions.

Once a UFrame is initialized and executed during the web session, user-owned resources (i.e., JavaScript heap objects of the UFrame) are isolated from the less-privileged code. These sensitive user-owned resources include credit card information, sensitive images, secret key information derived from the authentication process, and other sensitive data tied to a user. To ensure compatibility with the existing web application, the users should be able to interact with (e.g., view or input into) these resources. For example, bank account number is a sensitive user-owned resource and this needs to be displayed or entered by Alice when she checks her transaction history. USERPATH introduces a set of secure DOM APIs (Table 2) to create secure input elements (e.g., textboxes, textareas) and secure display elements (e.g., images and styled-texts). Secure elements are akin to standard HTML input and display elements, except that these elements are not accessible to scripts outside the UFrame on the web origin O. For instance, only event handlers (e.g., keyboard inputs and mouse clicks) inside the UFrame code can access the secure display and input elements, and these handlers cannot be overridden by code outside the UFrame. Therefore, USERPATH establishes a secure input and visual channel to safeguard sensitive display and input elements.

End-to-End Trusted Path. Finally, a UFrame needs to communicate back to the server. The main challenge is that the server needs to disambiguate HTTP requests generated by the UFrame in response to the authentic user interaction, as opposed to fake requests generated by malicious scripts via PISE attacks. USERPATH handles this issue by creating a dedicated network channel for the UFrame code. Inside the initialized UFrame code, the server embeds a set of nonces S called *user interaction token set* (Figure 5 Step C) that can be used to generate resource access HTTP requests from client side. These tokens can only be attached by the browser kernel as a custom HTTP header `X-UFRAME` when the UFrame-dedicated `XMLHttpRequest` is invoked (Step D).

Teardown. As the user Alice logs out of O, the server invalidates the session key K_s, and sets a custom HTTP header `X-USERPATH:Session-destroy` in HTTP response for the log out request (Figure 5 Step E). After getting this response, the browser destroys all user interaction tokens for the session and the session key K_s. To allow session reconnection, similar to cookies, the browser caches the user interaction tokens and K_s until the user logs out. The server then redirects the request to the login page if the key and tokens expire.

Table 2: Secure DOM APIs for UFrame

Downcall API	Description	Upcall API	Description
`createSecElement`	Create a secure UI element	`storeSecretKey`	Store the key K_s that is derived from PAKE protocol
`getSecElementById`	Get the secure UI element's object by ID	`updateUFrameCont`	Update the UFrame code or data content
`setSecElmAttr`	Set the property of an object with the corresponding value	`createContext`	Create a UFrame context that runs with user privilege. It lets the UFrame access privileged APIs
`getSecElmAttrVal`	Get the property's value of an object	`removeSecretKey`	Remove the secret key K_s during teardown process
`deletePAKESesKey`	Delete the session key K_s from the browser kernel	`removeUIToken`	Remove the interaction token T during teardown process

3.3 Security Properties: Putting it together

USERPATH enforces the following security semantics, which ensures resilience against PISE attacks.

- **P0**: **Safe Mutual Authentication & K_s Establishment.** Mutual authentication between user Alice and the server is required for web servers to securely delegate user Alice's authority O_{Alice} to client-side code within its web origin's authority O. This delegation is bootstrapped by Alice's user name and password. The secure delegation process must ensure that credential information does not leak outside Alice's authority, such as to attacker-controlled domains. After successful authentication, a session key K_s is derived. The key K_s must remain unforgeable, unguessable, and unique during the sessions.
- **P1**: **Secure Delegation.** A UFrame code that is passed from the backend server needs to be signed by K_s that is derived from mutual authentication between user and web server. Once web browser receives the content of the UFrame, it has to check the authenticity of the code with respect to K_s.
- **P2**: **Post-initialization Security of UFrame.** All sensitive data and code must be kept isolated inside a UFrame. The rest of the application code outside UFrame must not be able to access this data and code whatsoever.

The properties P0, P1 and P2 serve as the basis for subsequent security properties P3, P4, and P5 described as follows.

- **P3**: **Secure Visual and Input Channels for Users**
  Visual channel. We reuse the standard secure visual channel that requires display, intent, spatio-temporal, and pointer integrity to ensure *distinguishability* of secure UI elements from the non-secure ones. Secure UI elements cannot be obstructed or tampered with by untrusted code. Its elements should be able to display confidential information to users and not be accessible to the non-UFrame code. This has been explored in other research works [15, 16, 32] and is not part of our contributions.
  Input Channel. All keyboard inputs to secure input elements go directly to the browser. The confidentiality and integrity of input action should not be violated by untrusted scripts. The browser should be able to distinguish genuine user interactions from those mimicked by JavaScript code.

- **P4**: **Secure Browser ↔ UFrame Channel.** A privileged UFrame can communicate to the browser directly in order to create secure UI elements or to read contents in DOM objects securely with no possibility of interception from untrusted code. The confidentiality, integrity and authenticity of such communications are maintained by the browser.
- **P5**: **Secure UFrame ↔ Server Channel.** Web server should be able to distinguish requests generated from the authentic user interaction, and those that are not. The communications between the UFrame and the server are protected in their confidentiality and integrity.

Due to space constraints, we give a more thorough example-by-example security analysis in our technical report [20].

3.4 Compatibility & Usability Implications

Our mechanism can be easily extended to handle authentication via Single-Sign On (SSO). If the server O delegates authentication to an SSO provider S, a separate HTTPS connection is established from the browser to S. Thereafter, the credential input element uses the username and password to initiate the PAKE authentication with S. Upon successful completion, the browser obtains a shared key K_s with S, which is also communicated by S to O in a separate channel. O can create a server-side representation for Alice using K_s. The browser thus creates a UFrame with the authority of $Alice@S$, which can isolate $Alice@S$ from another user.

Usability Implications. First, we assume that web application users will always check the background color of any credential-seeking elements, and only enter their passwords if the color matches that of a rectangle displayed in the browser's chrome area. Second, we rely on prior research [15,16,32] to ensure the visual, temporal and pointer integrity of a secure visual channel. Admittedly, the usability of such a scheme has not been fully evaluated; a thorough user study on its usability merits separate research (c.f., [41,42]).

4 Implementation in Chromium

We summarize the high-level abstraction of our end-to-end solution and detail how it is implemented in Chromium web browser.

Implementation Overview. We implemented UFrame and trusted path components by modifying Chromium[5], the open source version of Google Chrome. We patched Chromium version 12 by adding roughly 475 lines of code spreading over 26 files inside Chromium codebase. This does not include the logic for performing PAKE procotol, which was implemented separately by us as a plug-in. Apart from the browser, we also modified 20 PHP-based server-side applications which we discuss in Section 5.

We have released our patch to Chromium and the modified web applications on a public repository [20]. We have also released a demo video showing how USERPATH offers smooth user experience with our running example Elgg [21].

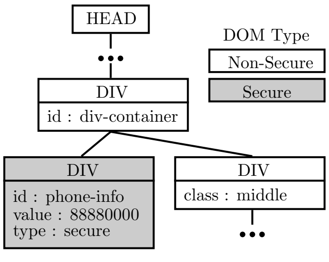
Authentication Step. As discussed in Section 3.2, once the browser identifies credential element on the HTML code, it renders this element and applies a random color on the element's background. To do so, we develop an NPAPI plug-in for the browser to

[5] http://www.chromium.org/

render such element and update the display color in web browser's chrome bar. As the credential element is rendered and called through privileged API, this is not accessible from web application code. To make the existing authentication process be UserPath-compliant, developers just need to embed the plug-in into original web application's login page.

```
<uframe
   src='http://www.example.com/content.php'
   sign='8d4f9a3112e700437e5cd783cc621'
   token='qvrz-clwo-xiud-jawz'>

// phone number obtained from the server
var jsonData = {"type":"display","elm":"div","
     id","phone-info","value":"88880000","
     parentNode":"div-container"};
// create secure div element append it into
     existing DOM element
var phoneElm = createSecElement(jsonData);
```



Internal DOM Representation

```
//-- Add a user as admin
var xhr = new XMLHttpRequest();
xhr.open('POST', 'http://'+URL+'/elgg/action/useradd', true);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');

var username = getSecElementById('username').getSecElmAttrVal('value');
var data = "username="+username;
...
// secure resource access to Elgg server
xhr.send(data);
</uframe>
```

Listing 1.1: Trusted Code Running in a UFrame. This piece of code executes under the user's authority O_{Alice} to create a secure `div` element into the web page and secure HTTP request to add a user as an admin. Details elided for brevity.

Subsequently, we employ the PAKE protocol to mutually authenticate user and the backend server by integrating TLS-SRP [43] — a PAKE-based web authentication that operates at the transport layer — into UserPath. On the web browser, we install a browser level TLS-SRP module that receives input from special credential box and carries out PAKE protocol with the specific origin O specified in `url` property of the UFrame code. The module consists of 381 C++ lines of code in total, which is roughly 2.6 MB in size. At the server side, we apply a patch to the Apache web server to handle server-side TLS-SRP authentication. This patch is available online [44].

Secure Delegation. After the authentication step finishes, the browser creates a UFrame for executing trusted JavaScript code. In this step, the browser already has a shared key K_s that can be used to secure communications with the server. Server-side web application then signs the content of the UFrame using the key K_s and sends it to the browser, embedded in a custom HTML tag named `<UFRAME>`. Whenever the browser encounters the UFrame content during parsing, it checks the integrity and authenticity of the UFrame code, and creates an `iframe` with a random origin $O_R = PRG(K_s)$, where $PRG(K_S)$ is a pseudorandom generator function that takes the shared key K_s as the seed.

We leverage existing mechanisms in the Chromium web browser to establish trusted paths. For ease of implementation, we modify *isolated worlds* [36], a feature provided by Chromium to separate execution context between two JavaScript code. This abstraction offers similar isolation mechanism as what `iframe`-based isolation with random origin provides.

Trusted Path Implementation. We use our running example in Section 2 to illustrate how we implement the trusted path execution inside a UFrame. As shown in Listing 1.1, UFrame code is purely written in JavaScript, and it has additional access to secure DOM APIs. As an example, we label contact information as a sensitive element to prevent them from being leaked to malicious code running on a web page. In Listing 1.1 line 9, a secure DOM element is created by invoking a downcall API `createSecElement()`. This API receives a JSON object `jsonData` as an input, and creates a secure display element based on data from `jsonData`. The object `jsonData` has user-owned contact information, which is sensitive data passed from the backend server to the browser. In Listing 1.1 line 10-19, we create a POST request directly from the UFrame using dedicated `XMLHttpRequest` to protect the client-side request to Elgg server. The data that is sent through the POST request (e.g., username, password) is obtained from user input on the secure input elements (Listing 1.1 line 15). As the `XMLHttpRequest` object is being called from UFrame, the browser treats the request as secure resource access to the server and appends special user interaction token for that request.

In our Chromium implementation, we make small changes in the following C++ classes: `ScriptController`, `V8IsolatedContext` and `V8NodeCustom`. We add a new data structure called `IsolatedContextMap` to maintain the relation between code running on the web page or the UFrame, represented by a context identifier. Therefore, the system can recognize the context where a JavaScript code is running by checking the data structure. Finally, we modify Chromium to mediate access from a JavaScript object to a DOM Node. The logic for mediating access to sensitive DOM element is as follows: as each element of the DOM is represented by an object, we add a special flag for every object that is created under specific privileged functions. We then modify the logic for traversing an object in a DOM tree, so that those objects with privileged flag will not be visible to the web application code running under origin O.

5 Evaluation

We deploy USERPATH on 20 open source web applications (as Table 4 shows) from 8 different categories (as Table 5 presents) including 3 frameworks (WordPress, Joomla, and Drupal). These web applications are statistically popular, built using PHP, and cover a wide range of functionalities. We evaluate our solution from four aspects – scope of vulnerabilities USERPATH can eliminate, case study of elgg, applicability to web applications & TCB reduction, and USERPATH's performance.

5.1 Scope of Vulnerabilities

We study a set of vulnerabilities in the web applications that can lead to PISE attacks. Among the 20 open source web applications that we study, there are 325 vulnerabilities on those web applications that can be exploited to launch the attacks. Most of them

Table 3: List of Vulnerabilities in 20 Open-source Web Applications. These vulnerabilities might lead to PISE Attacks

App Name & Version	Popularity Indicator	PHP # of LOC	Sensitive User Data	# of Relevant Vulnerabilities
Elgg v1.8.16	>2,800,000 downloads	114735	Private profile data and admin options (set user as admin and add new user)	**3** (CVE-2012-6561: XSS, EDB-ID 17685 and 8993: XSS)
Friendica v3.2.1744	Forbes's Top 3 social network application	144555	Private contact, friend list, and message data	**1** (Bug ID 0000535: Reflected XSS)
Roundcube v0.9.4	>2,400,000 downloads	109663	Address book, settings and private emails	**12** (CVE-2013-5646: XSS and CVE-2009-4077: CSRF)
OpenEMR v4.1.2	Serving >30,000,000 patients	495987	Personal info, medical records, and payment	**2** (ZSL-2013-5129 and 103810: XSS)
ownCloud v5.0.13	>350,000 users	337192	Contacts, export files and user share options	**15** (CVE-2013-1942: XSS and CVE-2012-4753: CSRF)
HotCRP v2.61	Used by USENIX, SIGCOMM, etc.	36333	Contact information, review and privilege settings	**3** (Bug ID 3f143d2: XSS)
OpenConf v5.30	Used by ACSAC, IEEE, W3C, ACM, etc.	17589	Contact info, review, edit submission and role setting	**1** (CVE-2005-0407: XSS and CVE-2012-1002: XSS)
PrestaShop v1.5.6.0	Powering >150,000 online stores	250660	Personal info, credit slips, addresses and checkout info	**2** (CVE-2008-6503 and CVE-2011-4544: XSS)
OpenCart v1.5.6	>250,000 downloads	93770	Account, address book and checkout information	**1** (CVE-2010-1610: CSRF)
AstroSpaces v1.1.1	DZineBlog's Top 10 open social network.	6972	Profile information, private message and admin settings	**1** (Bug ID 001: XSS)
Magento v1.8.0.0	Used by >200,000 business	928991	Account info, address information and checkout info	**1** (CVE-2009-0541: XSS)
Zen Cart v1.5.1	>3,000,000 downloads	95381	Account, profile and checkout information	**4** (CVE-2011-4567 and CVE-2012-1413: XSS)
osCommerce v2.3.3.4	>12,000 registered sites with >270,000 members	60081	Account, profile and checkout information	**10** (CVE-2012-1792 and CVE-2012-2935: XSS)
StoreSprite v7.24.4.13	Incorporate 14 payment gateways	30350	Account, profile and checkout information	**1** (CVE-2012-5798: XSS)
CubeCart v5.2.4	Powering thousands of online stores	11942	Account, profile and checkout information	**1** (CVE-2008-1550: XSS)
WordPress v3.6	Used by >60,000,000 websites	135540	Account, contact and setting information	**91** (CVE-2013-5738: XSS and CVE-2013-2205: XSS)
Joomla v3.2.0	>35,000,000 downloads	227351	Account, contact and setting information	**45** (CVE-2013-3059 and CVE-2013-3267: XSS)
Drupal v7.23	>1,000,000 downloads	43835	Account, contact and setting information	**126** (CVE-2012-0826: CSRF and CVE-2012-2339: XSS)
Piwigo v2.5.3	Translated into 50 languages	143144	User's management, permission, sensitive profile	**4** (CVE-2013-1468: CSRF and CVE-2012-2209: XSS)
X2CRM v3.5.6	>4,500 installations across 135 countries	747261	Account, contact management & information	**1** (CVE-2013-5693: XSS)

have been patched and recorded in the vulnerability database, but some of them are still unpatched.

Table 3 lists our case study and summarizes the number of vulnerabilities, along with the CVE ID for the corresponding vulnerability[6]. Among those 20 web applications that we study, all of them have at least one vulnerability to a subset of PISE attacks namely XSS or CSRF attacks. Some of them even have more than ten vulnerabilities of the same attack vector. To name one of them, PrestaShop has two critical vulnerabilities. One type of vulnerability (marked by ID CVE-2008-6503) allows an attacker to inject arbitrary web scripts to the login page. The other vulnerability (marked by ID CVE-

[6] Due to the page limit, we show the study of 8 applications from 8 different categories. For the study on all 20 applications, please check our technical report [20].

2011-4544) lets the attacker to exploit the file management process of an administrator to launch an XSS attack.

5.2 Case Study : Elgg & OpenCart

In this section, we detail our experience with real-world case studies to illustrate the steps taken for retrofitting web applications with USERPATH. We evaluate USERPATH with the following goals: (1) protecting the "add new user" feature in Elgg social network, given the presence of XSS vulnerabilities and (2) protecting the "reset password" feature in OpenCart, given a CSRF vulnerability in the web application. Based on those vulnerabilities, we thus construct four proof-of-concept attacks that tamper with the four channels discussed in Section 2.1. Due to space constraints, we describe the attacks and specify the way USERPATH prevents those attacks in our technical report [20].

Code Changes. First, we made small changes in `actions/login.php` (Elgg) and `account/login.php` (OpenCart) to let the browser render special credential boxes and initiate a TLS-SRP-based authentication with the server at their respective origins. Secondly, in the "add new user" page of Elgg, we privilege-separated the logic for displaying username, email address, password, admin flag, and a form request button into a `UFrame` section. Instead of creating those elements using HTML, the elements need to be dynamically created from within a UFrame to let them be rendered as secure elements. All the changes were made in a PHP file `forms/useradd.php`. Likewise, we protect two HTML input elements for putting in new password and a confirmation button by implementing the logic for this feature separately inside a `UFrame`. All these changes were made by modifying a file `account/password.php`. A complete set of technical changes is described in [20].

Result & Challenges. We successfully retrofitted USERPATH to Elgg and OpenCart by adding 270 and 266 lines of PHP code in their application code, respectively. The TCB size of the UFrame in the modified Elgg is 46x and 66x smaller than the size of TCB in vanilla web applications. After implementing those changes, we successfully protect the sensitive resources in the vulnerable applications from PISE attacks. We demonstrate some of the attacks in Elgg and how USERPATH defends against those through demo videos available in [21].

The main challenge of adopting USERPATH to web applications is the difficulty in locating the functionality we need to modify, because both applications were built using their own toolkit. After understanding the toolkit, the modification effort is straightforward. It took 2 days in total for us to enable USERPATH in Elgg and OpenCart.

5.3 Applicability to Web Applications & TCB Reduction

We successfully retrofit all 20 web applications to adopt USERPATH. Among these applications, we manually choose several data and operations that are sensitive to users (summarized in Table 3) and modify the PHP files where these data and operations are processed. In addition, we demonstrate the practicality of USERPATH by summarizing the adoption effort and TCB reduction of 20 retrofitted web applications in Table 4. We measure the adoption effort by the following benchmarks: number of additional code, number of modified files, and number of days spent in modifying the web application. Besides, we also measure TCB reduction by comparing the initial TCB size (i.e., the web page size) and the final TCB size after implementing USERPATH.

Table 4: Adoption Effort and TCB Reduction after Implementing UserPath in 20 Open-Source Web Applications

App Name	UserPath LOC (JS+PHP)	Original TCB (KB)	TCB after implementing UserPath (KB)	TCB Reduction Factor	# of Modified Files	# of Days Spent
Elgg	270	414.6	9.1	46x	4	2
Friendica	176	1053.8	5.3	199x	13	1
Roundcube	96	946.0	8.0	118x	4	2
OpenEMR	141	53.6	6.6	8x	7	1.5
ownCloud	106	555.2	2.9	191x	4	1.5
HotCRP	139	184.5	4.6	40x	5	1
OpenConf	151	55.9	2.4	23x	5	1
PrestaShop	111	580.3	5.8	100x	5	1
OpenCart	266	754.8	11.5	66x	6	2
AstroSpaces	119	67.3	3.5	19x	5	1
Magento	227	987.0	11.2	88x	4	1.5
Zencart	130	241.8	6.5	37x	6	1
osCommerce	122	425.8	5.9	72x	5	1
StoreSprite	133	513.8	4.6	112x	4	1
CubeCart	118	469.2	6.2	76x	5	1
WordPress	102	308.7	3.9	79x	4	1
Joomla	87	819.3	3.1	264x	3	1
Drupal	72	199.6	2.6	77x	3	1.5
Piwigo	216	673.5	7.8	86x	6	1
X2CRM	217	1380.4	6.1	226x	10	2

We find that UserPath requires small changes to the existing web application code. Given the set of sensitive user-owned data and functionalities that we want to protect from PISE attacks, we only need to add at most 270 lines of PHP and JavaScript code into the web application, with 167 lines of code added for each web application on an average (see column "UserPath LOC" in Table 4 for LOC of all the 20 applications). Moreover, we empirically show that we achieve the reduction of 8x to 264x in TCB for our case studies. We measure this reduction by comparing the size of final TCB (e.g., the UFrame code) with the entire web page size (see column IV in Table 4). We treat the web page size as the initial TCB size as we need to trust the entire web page in order to protect our sensitive data and operation.

We also find that modifying web applications according to UserPath incurs relatively small burden on the developer side. On the average, given a set of sensitive user-owned resources to protect in Table 3, a developer needs to modify 6 files within 1.3 days for one web application to make it UserPath-compliant.

5.4 Performance

The main performance factor that impact our solution include: PAKE-based secure delegation, the UFrame creation, and new secure elements introduced into DOM. As our demo video [21] shows, in our experiments with the 20 web applications, we do not observe any slowdown in user interactions with the applications. Since the login phase contains all the three factors, we measure the overhead of the login time for 20 applications from 8 different categories. Table 5 summarizes the results of the login time (averaged on 5 runs) between the click on the login button and the next page finishes loading. We can see that UserPath introduces the negligible performance overhead to these applications. This confirms our speculation that the minimal performance overhead that might incur from UserPath would be largely masked by the timing variances in network requests.

Table 5: Time Taken for Login without & with UserPath (in seconds)

Category	Application Name	Time without UserPath	Time with UserPath	Overhead
Social Networking	Elgg	3.38	3.45	2.07%
Social Networking	Friendica	4.88	5.02	2.87%
Social Networking	AstroSpaces	0.397	0.406	2.27%
Email Application	Roundcube	7.28	7.49	2.88%
Health Information System	OpenEMR	3.238	3.338	3.09%
Conference Management System	HotCRP	1.037	1.065	2.70%
Conference Management System	OpenConf	0.173	0.176	1.73%
E-commerce Application	OpenCart	4.26	4.40	3.29%
E-commerce Application	PrestaShop	3.52	3.56	1.14%
E-commerce Application	Magento	3.02	3.07	1.66%
E-commerce Application	Zencart	1.16	1.2	2.83%
E-commerce Application	osCommerce	7.38	7.46	1.08%
E-commerce Application	StoreSprite	5.03	5.13	1.99%
E-commerce Application	CubeCart	3.05	3.09	1.31%
Content Management System	WordPress	3.708	3.777	1.86%
Content Management System	Joomla	2.74	2.81	2.55%
Content Management System	Drupal	1.56	1.62	3.44%
File Sharing System	Piwigo	1.55	1.57	1.09%
File Sharing System	ownCloud	5.2	5.36	3.08%
Customer Management System	X2CRM	9.105	9.364	2.84%

6 Related Work

In this section, we discuss recent research works that are related to our solution.

Privilege Separation. Privilege separation reduces the potential damages of compromised software components by partitioning software into different compartments. It has been widely adopted in traditional applications [45,46], web browsers [47–49], and web applications [18, 34]. View isolation implemented by PathCutter [24] separates code running in different `iframes` (views) as well as requests coming out of different views. Thus, it prevents unwanted access to data between views, either directly or indirectly via sending requests to the server. Our solution in this paper applies privilege separation using a user-centric approach. We bring in user sub-origins to the present web, and confine user data only to code delegated by the user sub-origin.

Data Confinement. Confining data in web applications has recently received attention in the research community. For instance, Roesner et al. propose ACG, which allows users to directly grant access to user-owned resources by UI interaction with such gadgets [15]. Our solution shares the similar insight as to confine user data back to user-sanctioned operations, although we face different challenges in protecting user data on the web. Unlike resources on OS, the distributed nature of the web and decoupled server-client architecture requires additional secure channels to confine user data on the web. We address such challenges by integrating TLS-SRP into web authentication to build an end-to-end trusted path from the client-side application code to the web server.

Several other works have been proposed to confine sensitive data on the web [8] or cloud platform [50]. Compared to these proposals, our solution does not confine user data according to any application-specific configuration or data propagation policies; instead, it ensures that user data only flows within user sub-origin, both at the client and the server side.

Trusted Paths. Building trusted paths across untrusted components has practical significance today. Prior works examine potential solutions for trusted paths between user-interaction elements and software applications [41, 51, 52]. Similarly, Web Wallet redesigns browser's user interfaces to protect user credentials against phishing attacks [29]. The usability of trusted path proposals has been evaluated in real-world usage [42, 51]. Zhou et al. propose a hypervisor-based general-purpose trusted path design on commodity x86 computers, and present a case study on user-oriented trusted path [53].

Our solution builds an end-to-end trusted path by utilizing the existing functionality of the web browser and server. This trusted path connects the user at the client side to the server, ensuring that only user-delegated sub-origins can access protected data. Such a trusted path differs from a recent proposal on a trusted path between user keyboard inputs and the web server, where no explicit notion of users is established [9]. Moreover, compared to it, our solution requires much smaller changes to web browsers; by piggybacking on passwords for authentication, we avoid the usability challenges in requiring users to generate and upload SSL keys as in [9]. Dong et al. propose a solution to identify requests crafted by injected scripts from those triggered by user interactions [13]. We apply a similar mechanism in our solution as part of input channel protection. However, their work focuses on monitoring and diagnosing web application behavior, and does not yield a solution for protecting data in web applications.

Injection Attack Prevention. As we discuss in this paper, injected scripts pose major threats to web applications. Previous endeavors of security researchers have devised numerous solutions to prevent or mitigate script injection, such as CSP [1], blueprint [54], DSI [55], and Noncespaces [56]. Nevertheless, in practice, it is difficult to eliminate all script injection vectors [2]. Our solution complement these solutions on script injection prevention as a second line of defense.

7 Conclusion & Acknowledgments

In this paper, we propose new abstractions to bring in the explicit notion of user sub-origins into the present web and establish an end-to-end trusted path between the user and the web server. We show that our solution eliminates a large amount of PISE attacks in real-world applications, and can be integrated with today's web browsers and applications with minimal adoption cost.

Acknowledgments. We thank the anonymous reviewers and our shepherd William Robertson for their feedback and suggested improvements for this work. We thank Kailas Patil, Atul Sadhu, Loi Luu, and Shweta Shinde for their comments on an early presentation of this work. This work is supported by the Ministry of Education, Singapore under Grant No. R-252-000-495-133. Xinshu Dong is supported by the research grant for the Human Sixth Sense Programme at the Advanced Digital Sciences Center from Singapore's Agency for Science, Technology and Research (A*STAR).

References

1. W3C: Content security policy 1.0. `http://www.w3.org/TR/CSP/`
2. Johns, M.: Preparedjs: Secure script-templates for javascript. In: Detection of Intrusions and Malware & Vulnerability Assessment. (2013)
3. Chen, P., Nikiforakis, N., Huygens, C., Desmet, L.: A dangerous mix: Large-scale analysis of mixed-content websites. In: Information Security Conference. (2013)

4. Trend Micro: New york times pushes fake av malvertisement. http://goo.gl/BtjgPc
5. Verizon: 2013 data breach investigation report. http://www.verizonenterprise.com/DBIR/2013/
6. Enigma Group: Facebook profiles can be hijacked by chrome extensions malware. http://underurhat.com/hacking
7. Liu, L., Zhang, X., Yan, G., Chen, S.: Chrome extensions: Threat analysis and countermeasures. In: Network and Distributed System Security Symposium. (2012)
8. Akhawe, D., Li, F., He, W., Saxena, P., Song, D.: Data-confined html5 applications. In: European Symposium on Research in Computer Security. (2013)
9. Dong, X., Chen, Z., Siadati, H., Tople, S., Saxena, P., Liang, Z.: Protecting sensitive web content from client-side vulnerabilities with cryptons. In: Proceedings of the 20th ACM Conference on Computer and Communications Security. (2013)
10. Parno, B., McCune, J.M., Wendlandt, D., Andersen, D.G., Perrig, A.: Clamp: Practical prevention of large-scale data leaks. In: IEEE Symposium on Security and Privacy. (2009)
11. Felt, A.P., Finifter, M., Weinberger, J., Wagner, D.: Diesel: applying privilege separation to database access. In: ACM Symposium on Information, Computer and Communications Security. (2011)
12. Chen, E.Y., Gorbaty, S., Singhal, A., Jackson, C.: Self-exfiltration: The dangers of browser-enforced information flow control. In: Web 2.0 Security and Privacy. (2012)
13. Dong, X., Patil, K., Mao, J., Liang, Z.: A comprehensive client-side behavior model for diagnosing attacks in ajax applications. In: ICECCS. (2013)
14. Projects, T.C.: Per-page suborigins. http://goo.gl/PoH5pY
15. Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H.J., Cowan, C.: User-driven access control: Rethinking permission granting in modern operating systems. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. (2012)
16. Roesner, F., Fogarty, J., Kohno, T.: User interface toolkit mechanisms for securing interface elements. In: User Interface Software and Technology. (2012)
17. Dong, X., Hong, H., Liang, Z., Saxena, P.: A quantitative evaluation of privilege separation in web browser designs. In: European Symposium on Research in Computer Security. (2013)
18. Akhawe, D., Saxena, P., Song, D.: Privilege separation in html5 applications. In: USENIX Security. (2012)
19. Oiwa, Y., Takagi, H., Watanabe, H., Suzuki, H.: Pake-based mutual http authentication for preventing phishing attacks. In: World Wide Web Conference. (2009)
20. Budianto, E., Jia, Y.: Summary of source code modification, chromium patches, and userpath technical report. https://github.com/ebudianto/UserPath
21. Budianto, E., Jia, Y.: Url for UserPath demo video. https://github.com/ebudianto/UserPath/wiki/DEMO-Video-URLs
22. Dietz, M., Czeskis, A., Balfanz, D., Wallach, D.S.: Origin-bound certificates: A fresh approach to strong client authentication for the web. In: USENIX Security. (2012)
23. Jackson, C., Simon, D.R., Tan, D.S., Barth, A.: An evaluation of extended validation and picture-in-picture phishing attacks. In: Proceedings of 1st USEC. (2007)
24. Cao, Y., Yegneswaran, V., Porras, P., Chen, Y.: Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In: Network and Distributed System Security Symposium. (2012)
25. Hansen, R., Grossman, J.: Clickjacking. http://goo.gl/p7dxIC
26. YGN Ethical Hacker Group: Elgg 1.7.9 xss vulnerability. http://goo.gl/XUeqis
27. CVE: Cve-2012-6561 xss vulnerability in elgg. http://goo.gl/mmW8bM
28. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: Conference on Computer and Communications Security. (2008)
29. Wu, M., Miller, R.C., Little, G.: Web wallet: Preventing phishing attacks by revealing user intentions. In: Symposium On Usable Privacy and Security. (2006)

30. Bhargavan, K., Delignat-Lavaud, A., Maffeis, S.: Language-based defenses against untrusted browser origins. In: USENIX Security. (2013)
31. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web application. In: IEEE Symposium on Security and Privacy. (2010)
32. Huang, L.S., Moshchuk, A., Wang, H.J., Schechter, S., Jackson, C.: Clickjacking: attacks and defenses. In: USENIX Security. (2012)
33. Zhou, Y., Evans, D.: Protecting private web content from embedded scripts. In: European Symposium on Research in Computer Security. (2011)
34. Dong, X., Tran, M., Liang, Z., Jiang, X.: Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In: ACSAC. (2011)
35. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: Computer Security Foundations. (2010)
36. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: Network and Distributed System Security Symposium. (2010)
37. Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., Venkatakrishnan, V.N.: Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In: Conference on Computer and Communications Security. (2010)
38. Wu, T.: The secure remote password protocol. In: Network and Distributed System Security Symposium. (1998)
39. The Spanner: Dom clobbering. http://goo.gl/ZOLmal
40. Adida, B., Barth, A., Jackson, C.: Rootkits for javascript environments. In: WOOT. (2009)
41. Ye, Z.E., Smith, S.: Trusted paths for browsers. In: USENIX Security. (2002)
42. Libonati, A., McCune, J.M., Reiter, M.K.: Usability testing a malware-resistant input mechanism. In: Network and Distributed System Security Symposium. (2011)
43. Engler, J., Karlof, C., Shi, E., Song, D.: Is it too late for pake? In: Proceedings of Web 2.0 Security and Privacy 2009
44. Slack, Q.: Tls-srp in apache mod_ssl. http://goo.gl/cHMoau
45. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: USENIX Security. (2003)
46. Brumley, D., Song, D.: Privtrans: automatically partitioning programs for privilege separation. In: USENIX Security. (2004)
47. Grier, C., Tang, S., King, S.: Designing and implementing the op and op2 web browsers. ACM Transactions on the Web (2011)
48. Wang, H.J., Grier, C., Moshchuk, A., King, S.T., Choudhury, P., Venter, H.: The multi-principal os construction of the gazelle web browser. In: USENIX Security. (2009)
49. Barth, A., Jackson, C., Reis, C., Team, T.G.C.: The security architecture of the chromium browser. http://goo.gl/BGjJqC
50. Papagiannis, I., Pietzuch, P.: Cloudfilter: practical control of sensitive data propagation to the cloud. In: Cloud Computing Security Workshop. (2012)
51. Tong, T., Evans, D.: Guardroid: A trusted path for password entry. In: MoST. (2013)
52. McCune, J.M., Perrig, A., Reiter, M.K.: Safe passage for passwords and other sensitive data. In: Network and Distributed System Security Symposium. (2009)
53. Zhou, Z., Gligor, V.D., Newsome, J., McCune, J.M.: Building verifiable trusted path on commodity x86 computers. In: IEEE Symposium on Security and Privacy. (2012)
54. Ter Louw, M., Venkatakrishnan, V.N.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: IEEE Symposium on Security and Privacy. (2009)
55. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: Network and Distributed System Security Symposium. (2009)
56. Gundy, M.V., Chen, H.: Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In: Network and Distributed System Security Symposium. (2009)